Chapter 8
Technical Debt

In this chapter I discuss the concept of technical debt. I begin by defining technical debt, which encompasses naive debt, unavoidable debt, and deliberate debt. Next I examine some common causes of technical debt and the consequences of accruing high levels of debt. I then describe three activities associated with technical debt: managing the accrual of technical debt, making technical debt visible, and servicing technical debt. I specifically emphasize how to apply these activities when using Scrum.

Overview

Ward Cunningham was the first to write about the concept of **technical debt** (Cunningham 1992). He defined it as follows:

> *Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. . . . The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation. . . .*

Cunningham used the technical debt metaphor to explain to his business team why creating software fast to get feedback was a good thing. In doing so, however, he emphasized two key points: The team and organization need to be vigilant about repayment of the debt as their understanding of the business domain improves, and the design and implementation of the system need to evolve to better embrace that understanding.

Since the introduction of the term in the early 1990s, the software industry has taken some liberties with Cunningham's definition. Nowadays, technical debt refers both to the shortcuts we purposely take and also to the many bad things that plague software systems. These include

- Unfit (bad) design—a design that once made sense but no longer does, given important changes to the business or technologies we now use
- Defects—known problems in the software that we haven't yet invested time in removing
- Insufficient test coverage—areas where we know we should do more testing but don't
- Excessive manual testing—testing by hand when we really should have automated tests

- Poor integration and release management—performing these activities in a manner that is time-consuming and error-prone
- Lack of platform experience—for example, we have mainframe applications written in COBOL but we don't have many experienced COBOL programmers around anymore
- And many more, because the term *technical debt* today is really used as a placeholder for a multidimensional problem

Cunningham didn't intend for technical debt to refer to team member or business immaturity or process deficiencies that lead to sloppy design, poor engineering practices, and a lack of testing. This kind of debt can be eliminated through proper training, a good understanding of how to apply technical practices, and sound business decision making. Because of the irresponsible and frequently accidental nature of how this type of debt is generated, I refer to it as **naive technical debt**. It is also known by other names: *reckless debt* (Fowler 2009), *unintentional debt* (McConnell 2007), and *mess* (Martin 2008).

In addition, there is **unavoidable technical debt**, which is usually unpredictable and unpreventable. For example, our understanding of what makes for a good design emerges from doing the design work and building user-valuable features on it. We can't perfectly predict up front how our product and its design will need to evolve over time. So, design and implementation decisions we made early on might need to change as we close important learning loops and acquire validated learning. The changes required in the affected areas are unavoidable technical debt.

As another example, say we licensed a third-party component for use in our product and the interfaces to that component evolve over time. Our product that once functioned well with the third-party component accrues technical debt through no fault of our own. Although this debt might be predictable (it's not unreasonable to assume that the vendor will change its component interfaces over time), it's not preventable because we can't foresee how the component developers might evolve the component in the future.

The final type of technical debt is **strategic technical debt**. This kind of debt is a tool that can be used to help organizations better quantify and leverage the economics of important, often time-sensitive, decisions. For example, an organization might deliberately make a strategic decision to take shortcuts during product development to achieve an important short-term goal, such as getting a time-sensitive product into the marketplace. Also, for a capital-strapped organization that is at risk of running out of money before it can complete its product, getting a product with technical debt to market at a reduced initial development cost and then generating revenue to self-fund ongoing development may be the only way for the organization to avoid death before deployment.

Regardless of how the debt was accrued, technical debt is a powerful metaphor because it raises awareness of and provides visibility into an important issue. The metaphor resonates well with business people who tend to be well versed in financial debt. When they hear *technical debt*, they can quickly appreciate the

insightful parallels, the most important being that just like financial debt, technical debt requires interest payments, which come in the form of extra future development effort. We can choose to continue paying the interest (by working around the problems), or we can pay down the debt principal (for example, by **refactoring** the code to make it cleaner and easier to modify).

Consequences of Technical Debt

As the level of technical debt rises, so does the severity of the consequences. Let's discuss a few of the more notable consequences of high levels of technical debt (summarized in Figure 8.1).

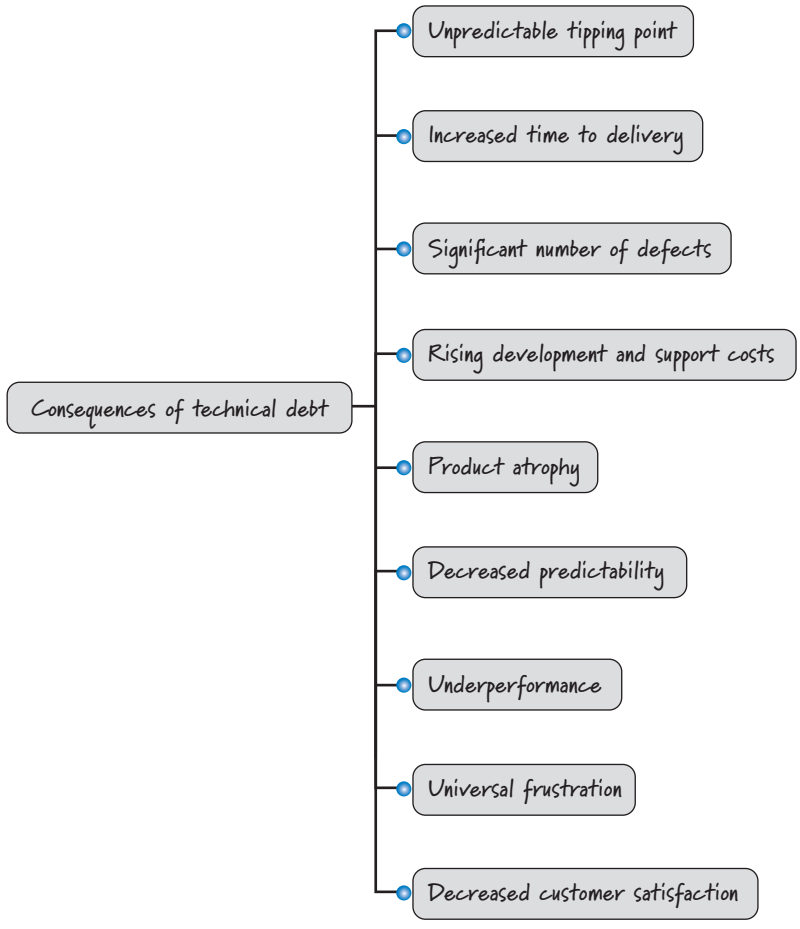

FIGURE 8.1 Consequences of technical debt

Unpredictable Tipping Point

An important attribute of technical debt is that it grows in an unpredictable, nonlinear fashion. Each bit of technical debt, when added to the pool of existing technical debt, might do significantly more harm than the size of that new debt might imply. At some point, technical debt achieves a sort of "critical mass," where the product reaches a tipping point and becomes unmanageable or chaotic. At the tipping point, even small changes to the product become major occasions of uncertainty. This nonlinear characteristic is a significant business risk; we don't know when the next piece of straw is going to break the camel's back, but when it does, all consequences are amplified.

Increased Time to Delivery

Taking on technical debt means taking a loan today against the time required to do future work. The greater the debt today, the slower the velocity tomorrow. When velocity slows, it takes longer to deliver new features and product fixes to customers. So, in the presence of high technical debt, the time between deliverables actually increases rather than decreases. In ever-competitive marketplaces, technical debt is actively working against our best interests.

Significant Number of Defects

Products with significant technical debt become more complex, making it harder to do things correctly. The compounding defects can cause critical product failures to happen with alarming frequency. These failures become a major disruption to the normal flow of value-added development work. In addition, the overhead of having to manage lots of defects eats into the time available to produce value-added features. At some point, we begin to drown but are so busy treading defect-filled waters we can't see how to pull ourselves out of the mess we are in.

Rising Development and Support Costs

As technical debt increases, development and support costs start rising. What used to be simple and cheap to do is now complicated and expensive. In the presence of increasing levels of technical debt, even small changes become very expensive (see Figure 8.2).

When the high technical debt curve in Figure 8.2 starts its aggressive climb, we reach a critical mass of technical debt and are at the tipping point.

Additionally, rising costs can change the economics of whether to proceed with a feature or defect repair. A feature that could be built (or a defect that could be repaired) at a low cost in the presence of low technical debt might become too expensive in the presence of high technical debt. As a result of rising costs, our products become less adaptive to the evolving environment in which they must exist.

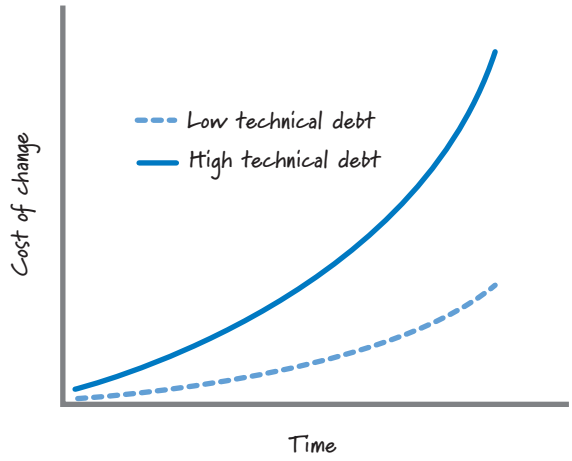

FIGURE 8.2 Cost-of-change curve affected by technical debt

Product Atrophy

As we stop adding new features or fixing defects that could rejuvenate our aging product, the product becomes less and less appealing to current and potential customers. As a result, the product starts to atrophy and simply ceases to be a viable option for most customers. Those who stay with the product are typically stuck with it for the time being. But as soon as the first opportunity to switch to another product appears, they'll probably take it!

Decreased Predictability

For a product with high levels of technical debt, making any sort of prediction is nearly impossible. For example, estimates become bad estimates even for the most experienced team members. There is simply too much uncertainty surrounding how long something might take when dealing with a debt-ridden product. Consequently, our ability to make commitments and have a reasonable expectation of meeting them is seriously impaired. The business stops trusting anything development has to say, and customers stop trusting anything the business has to say!

Underperformance

Sadly, as technical debt increases, people come to expect increasingly lower development performance and therefore reduce their expectations of what is possible. Of course, the lowered expectations start to propagate through the value chain, resulting in lower overall performance on an organization-wide basis.

Universal Frustration

The unfortunate human consequence of high technical debt is that everyone in the value chain becomes frustrated. The accumulation of all of those small but annoying shortcuts makes work on the product painful. Eventually the joy in development disappears and is replaced with the day-to-day grind of fighting issues that no one wants to (or should have to) deal with. People burn out. Knowledgeable members of the development team begin to leave to pursue more gratifying opportunities; and, as they are the ones in the best position to actually do something about the debt problem, their leaving makes things even worse for those who remain. Morale spirals downward with increasing intensity.

Technical debt doesn't suck the joy out of just technical people; it has the same effect on business people. How long do we want to keep making business commitments that can't be met? And what about our poor customers, who are trying to run their businesses on top of our debt-ridden product? They, too, quickly grow tired of the repeated product failures and our inability to fulfill any promises that we make. The trust that once existed through the value chain is replaced with frustration and resentment.

Decreased Customer Satisfaction

Customer satisfaction will decrease as customer frustration increases. So the extent of the damage caused by technical debt is not just isolated to the development team or even to the development organization as a whole. Even worse, the consequences of technical debt can substantially affect our customers and their perception of us.

Causes of Technical Debt

Recall that technical debt comes in three main forms, each of which has a different root cause. Unavoidable technical debt accrues regardless of the preventive measures we adopt. Naive technical debt results from team member, organizational, and/or process immaturity. Strategic debt is something we might choose to take on when the benefits of accruing the debt substantially exceed the cost of the debt.

Pressure to Meet a Deadline

Both strategic and naive technical debt, however, are often driven by business pressure to meet an important looming deadline (see Figure 8.3, based on Mar 2006).

The vertical dimension represents the amount of work we want to accomplish by a desired release date (shown on the horizontal dimension). The line between the amount of work and the desired release date represents the constant projected velocity at which work must be completed to meet the desired release date. By working at the projected velocity, we aim to complete high-quality features in a timely way while minimizing the accrual of technical debt.

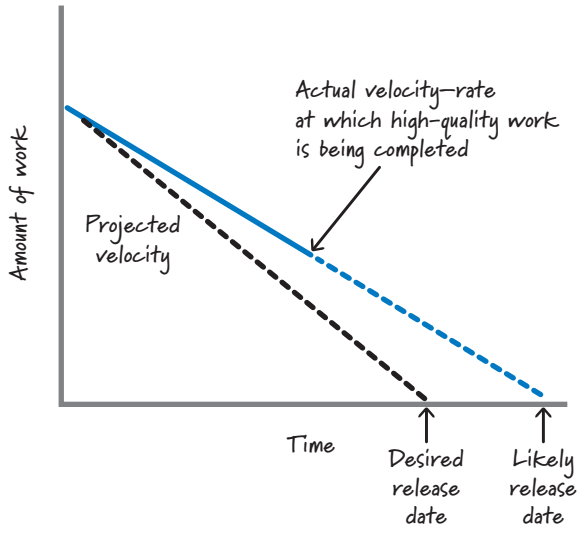

FIGURE 8.3 Pressure to meet a deadline can lead to technical debt.

However, as we start doing the work, the actual velocity needed to produce high-quality results is slower than the projected velocity. If we continue producing results at the actual velocity, we'll miss the desired release date and finish instead on the likely release date.

Attempting to Falsely Accelerate Velocity

At this point we need to make a business decision. Do we want to cut scope to meet the desired release date, or do we wish to add more time to the schedule to accommodate a delivery on the likely release date? Unfortunately, in many circumstances the business rejects both of those options and decrees that the team must meet the desired release date with all of the features. In this situation, the team doing the work is being told to accelerate its velocity to meet the desired release date (see Figure 8.4).

By working at this accelerated velocity, the team will have to make deliberate decisions to take on technical debt (meaning they will have to cut corners to work fast enough to meet the desired release date). Perhaps the design won't be as good as it should be, or specific types of testing (perhaps load testing) will be deferred. As a result, we will accrue technical debt as shown in the triangular region of Figure 8.4. This region represents all of the work we should have done but didn't have time to do.

Myth: Less Testing Can Accelerate Velocity

A prevalent myth is that testing is additional overhead, and by reducing it, we can accelerate velocity (see Figure 8.5).

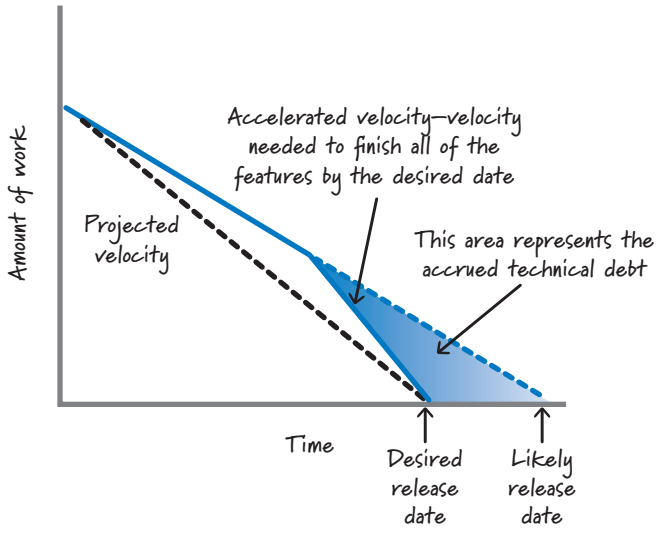

FIGURE 8.4 Accruing technical debt to meet unreasonable fixed scope and date

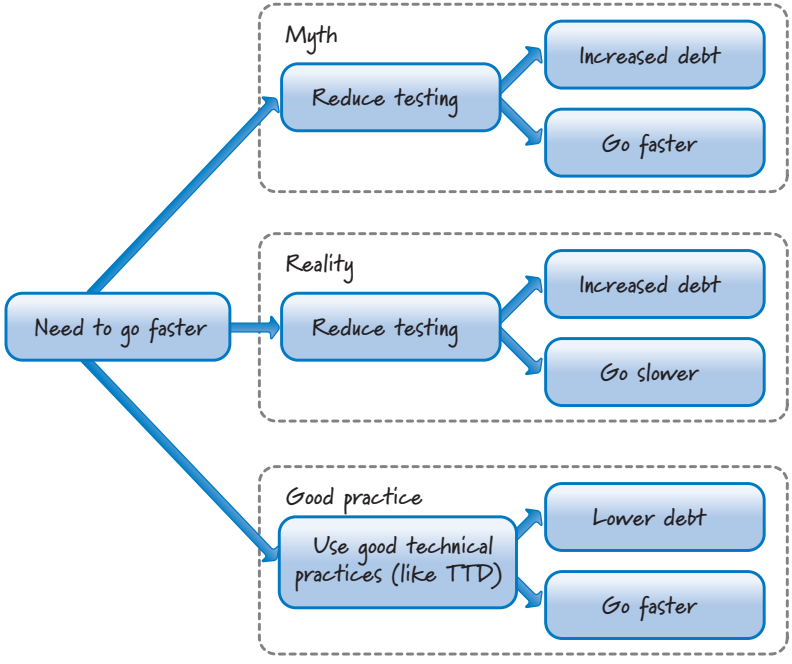

FIGURE 8.5 The myth, reality, and good practice of how testing affects velocity

The reality is that reducing testing will both increase debt and cause us to go slower, because problems will go undetected until later when it is much more time-consuming to fix them. Experienced teams deliver good-quality results faster and with less technical debt when testing is fundamentally integrated into the development process. These teams use good technical practices such as **test-driven development (TDD)**—where the developer writes and automates a small unit test before writing the small piece of code that will make the test pass (Crispin and Gregory 2009).

Debt Builds on Debt

Future technical debt builds quickly on top of existing technical debt. And, as the technical debt begins to build, economically harmful consequences start to appear. Figure 8.6 illustrates the consequences of building Release 2 on top of the technical debt from Release 1.

In Figure 8.6, the actual velocity during Release 2 is slower than it was in Release 1. It is clear that at this velocity we once again will miss the targeted release date. And, once again, the business insists that the team meet the desired release date with all of the features. As a result, we accrue even more technical debt.

If this pattern continues, eventually the velocity line might become horizontal. This would be a state where the technical debt in the system is so high that our effective velocity is zero. The result is the kind of product in which we are terrified to make any changes, because a small change in one area could cause 18 other things to break in what appear to be totally unrelated areas of the product. Worse yet, there is no way we can predict that those specific 18 things would break. And, of course, we

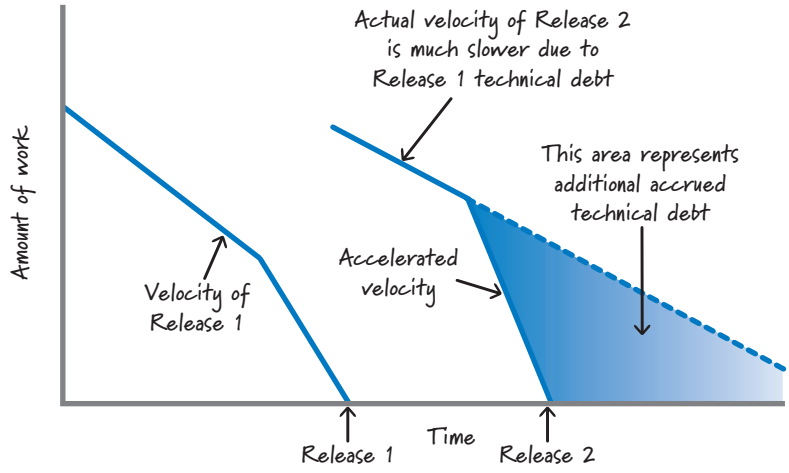

FIGURE 8.6 As technical debt increases, velocity decreases.

don't have any appreciable test framework to help us determine when they break—but, not to worry, our customers are sure to let us know!

Once we find ourselves in a situation with high technical debt, all choices become bad choices:

- Do nothing, and the problem gets worse.
- Make ever-larger investments in technical debt reduction that can consume more and more of our valuable product development resources.
- Declare technical bankruptcy, retire the technical debt, and replace the debt-ridden product with a new product at the full cost and risk of developing a new product.

With choices like these looming on the horizon, it is critical that we properly manage our technical debt before it spirals out of control.

Technical Debt Must Be Managed

Technical debt, like financial debt, has to be managed. It is important to realize that no product will be debt free, so I'm not suggesting that you try to achieve a debt-free status. Even if it were possible, the economics of being debt free simply might not be justified. We should, however, keep technical debt low enough that it doesn't significantly affect future product development.

Technical debt management requires a balanced technical and business discussion that must involve technical and business people. That is one reason why each Scrum team has a product owner. Having the product owner as part of the Scrum team allows for a balanced discussion of business and technical perspectives to make good economic trade-offs. As I will describe in Chapter 9, it is therefore essential that we choose a product owner with the proper business acumen to participate in these discussions.

There are three principal technical debt management activities (see Figure 8.7). I will address each of these activities in the following sections.

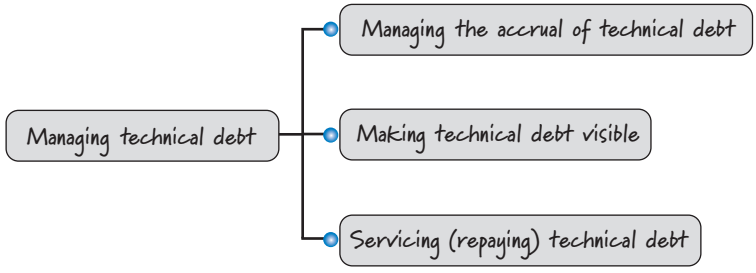

FIGURE 8.7 Activities for managing technical debt

Managing the Accrual of Technical Debt

A critical dimension of managing technical debt is to manage the debt accrual process. As I discussed earlier, there is only so much technical debt we can take on before we reach a critical mass. By analogy, continuously accruing technical debt is equivalent to continuously borrowing money against our house. At some point we just need to stop and say, "No more!" because the consequences become too severe.

First, we need to stop adding naive debt to our products (stop being reckless and creating messes). We also need to realize that there is only so much strategic debt or unavoidable debt we can accrue without repayment before we reach the tipping point. I will discuss approaches to addressing each of these. I won't discuss how to manage the accrual of unavoidable debt, because by its nature it is unpreventable (but we can make it visible and service it once we discover it).

Use Good Technical Practices

The first approach to managing the accrual of technical debt is to stop adding naive debt to our products. Using good technical practices is an excellent starting point. Although Scrum does not formally define **technical practices**, every successful Scrum team that I have seen employs practices such as simple design, test-driven development, **continuous integration**, automated testing, refactoring, and so on (see Chapter 20 for additional discussion). Understanding and proactively using these practices will help teams stop adding many forms of naive debt to their products.

In the case of accrued technical debt, code refactoring is an important tool for paying it down. Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior (Fowler et al. 1999). In other words, we clean up under the hood, but from the customer's perspective the product still works the same. By refactoring, we strive to reduce complexity while improving maintainability and extensibility. The result of refactoring is making the work at hand easier (the equivalent of reducing interest payments).

Cunningham (2011) explains the benefits of refactoring by example:

> *. . . customer is willing to pay for a new feature; feature doesn't fit in; reorganize the code so that it does fit in; now the feature is easy to implement. This could be called just-in-time refactoring. I would explain this to management as follows: we hope to have a place in our software for every new request. But sometimes we don't have a place for a feature so we have to make the place first, then implement the feature. . . .*

Use a Strong Definition of Done

Work that we should have performed when a feature was built, but ended up deferring until a later time, is an important cause of technical debt. Using Scrum, we want

a strong definition of done (see Chapter 4) to help guide the team to a low- or no-debt solution at the end of each sprint.

The more technically encompassing we make our definition-of-done checklist, the less likely we are to accrue technical debt. And, as I discussed in Chapter 2, many times the cost of paying back technical debt that slips past a weak definition of done is substantially greater than addressing it during the sprint. Operating without a strong definition of done is like granting a license to accrue technical debt.

Properly Understand Technical Debt Economics

To use technical debt strategically and advantageously, we must properly understand how it affects the economics of our decisions. Sadly, most organizations don't understand the implications of technical debt well enough to correctly quantify the economics of taking it on. Let me illustrate by example (see Figure 8.8).

In this example, assume the following:

- Each month of development costs $100K.
- We cannot reasonably meet the target delivery date (at ten months) with all of the requested, must-have features.
- Dropping features is just not an option.

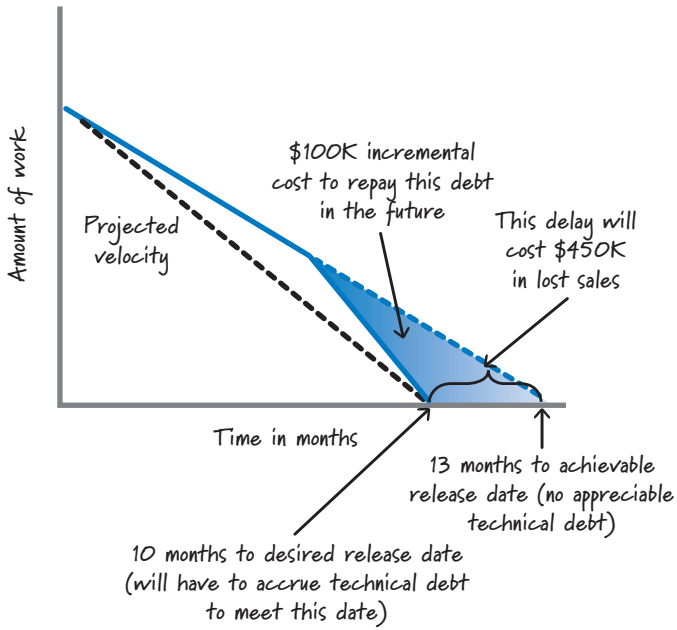

FIGURE 8.8 Example technical debt economic analysis

Let's consider two possible alternatives. First, delay the delivery date of the product by three months so that we can reasonably and professionally complete the work on the must-have feature set with minimum technical debt at 13 months. The total development cost would then be $1.3M. In discussions with sales and marketing, we also project that a three-month cost of delay equates to $450K in lost sales.

Second, accelerate development by taking shortcuts in order to meet the original target delivery date at ten months. To correctly quantify the economics of this option, we need to know the cost of taking on the technical debt.

This is where things get difficult. Imagine that we ask the development team, "So, if you have to make some design and implementation compromises today to get the must-have features done by the original desired date, how much additional money will it take to repay the debt after we do the first release?"

Let's say the team discusses the question and believes it will need four months to get the system cleaned up. This means the team will need one additional month over and above the three months it "saved" by originally cutting corners. The net result is that the team will spend an incremental $100K on development ($1.4M on development instead of the $1.3M in the first option). That's $100K the organization would not have to spend if we took the time to do the work the right way and did not put the technical debt into the product in the first place.

On the surface, the correct economic decision seems clear. Should we take on a technical debt of $100K to generate incremental revenue of $450K? Sure, who wouldn't do that? And that might be the correct answer if we believe that we have considered all (or even most) of the important cost factors associated with the technical debt.

However, here are just two of what could be many factors that we didn't consider:

- What about the delay cost of having to repay the technical debt? The $100K covers the expense of the team to do technical debt reduction work in the future. However, what about the cost of the time to do the debt reduction? Time spent repaying the principal on the debt is a delay cost on some other product or the next release of the same product. What is the cost of that delay? So, if it takes the team one extra month to repay the debt, some other product's release is likely delayed by one month. That lost opportunity cost has a real economic impact that must be considered.
- Most organizations are not good about repaying their technical debt. When push comes to shove, business people frequently favor developing new features versus reworking features that already exist. So, the reality is that we may not actually end up repaying any or all of the debt, which means we will likely have to pay interest on the debt for the useful life of the system. This also must be considered.

Table 8.1 summarizes the numbers of this example.

TABLE 8.1 Example Economics of Avoiding versus Taking on Technical Debt

	Avoid Debt	Take on Debt
Monthly development cost	$100K	$100K
Total development months	13	10
Total development cost	$1.3M	$1M
Delay in months (to release product)	3	0
Delay cost per month	$150K	$150K
Total delay cost	$450K	0
Debt-servicing months	0	4
Debt-servicing cost	$0	$400K
Total cost in lifecycle profits	$1.75M	$1.4M
Delay cost of incremental time to repay debt	$0	X
Lifetime interest payments on technical debt	$0	Y
Other debt-related costs	$0	Z
Real cost in lifecycle profits	$1.75M	$1.4M + X + Y + Z

Clearly, technical debt has tentacles that reach out and affect many different aspects of the overall economic calculation. Failing to consider at least the most important of these factors will ensure that we won't correctly quantify the economics of assuming technical debt.

Of course, if the economics in favor of taking on the debt are overwhelming and compelling—for example, we will go out of business if we don't take on that debt and get the product into the marketplace with all of the must-have features, or we will miss being first to market and lose the lion's share of the marketplace revenue—we don't need to spend time considering less important factors because we already know it's economically sensible to take on the debt.

More often, however, the decision isn't so clear-cut. The choice of whether or not to assume the debt usually requires detailed analysis to discern which is the better option. When deciding, err on the side of not taking on the debt. In my experience, most organizations substantially underestimate the true cost of assuming technical debt and aren't nearly as diligent as they think they will be at repaying it.

Making Technical Debt Visible

One of the principal benefits of the technical debt metaphor is that it enables the development team and the business people to have a necessary conversation using a shared context. To have that conversation, both need visibility into the product's technical debt position in a way that each can understand.

Make Technical Debt Visible at the Business Level

The problem in many organizations is that whereas the development team has at least some reasonable visibility into the product's technical debt position, the business people typically do not. Ask any technical person who has knowledge of a product where the greatest concentration of technical debt in the product is, and chances are she can answer that question. Ask the same question of a business person and she will typically have no appreciable understanding of how much, or what type of, technical debt exists.

The same would not be true for financial debt. Ask a business person about the organization's financial debt position and she will be able to give you a very accurate answer.

So it is essential to provide business people with visibility into the product's technical debt position. If I could quantify technical debt numerically—and there is significant current research work in the area of how to quantify technical debt (SEI 2011)—I might consider entering short-term and long-term technical debt line items on the organization's balance sheet right next to financial debt (see Table 8.2).

I can't actually point to any organizations that have short-term and long-term technical debt items on their balance sheets (although I think it is a good idea). I am

TABLE 8.2 Technical Debt Shown on the Organization's Balance Sheet

Assets		Liabilities	
Cash	$600K	Current Liabilities	
Accounts Receivable	$450K	Notes Payable	$100K
		Accounts Payable	$75K
		Short-Term Technical Debt	**$90K**
Tools and Equipment	$250K	Long-Term Liabilities	
		Notes Payable	$300K
		Long-Term Technical Debt	**$650K**
. . .	. . .	. . .	. . .

simply using this as an example to illustrate that each organization needs to find a way to communicate the magnitude of a product's technical debt in a way that the business people can understand. Otherwise, the business doesn't have the proper visibility into the true condition of the product to make informed economic decisions.

A way that some organizations do make visible the business consequences of technical debt is by tracking velocity over time. Figure 8.6 illustrated how an increase in technical debt results in a decrease in velocity. This decrease can be described in financial terms. For example, assume we have a Scrum team with a fixed cost per sprint of $20K and a historical velocity of 20 points per sprint. Using these numbers, we can compute that the team has a cost per point of $1K. If the accrual of technical debt causes the team's velocity to decrease to 10 points per sprint, the cost per point will rise to $2K. In aggregate, if the team has roughly 200 points of work to complete and velocity declines by one-half, what would have cost $200K to complete will now cost $400K. So, using velocity, we can clearly see the financial cost of the interest payments on the accrued technical debt.

Make Technical Debt Visible at the Technical Level

Technical people often have **tacit knowledge** of where at least the most egregious technical debt is located in the product. However, that understanding may not be visible in a way that it can be analyzed, discussed, and acted upon. Figure 8.9 illustrates three ways of making technical debt visible at the technical level.

First, technical debt could be logged like defects into an existing defect-tracking system (left side of Figure 8.9). This has the advantage of putting the debt in a familiar place using known tools and techniques. If the debt information is colocated with defect information, it is important to tag the debt in a way that it can easily be found,

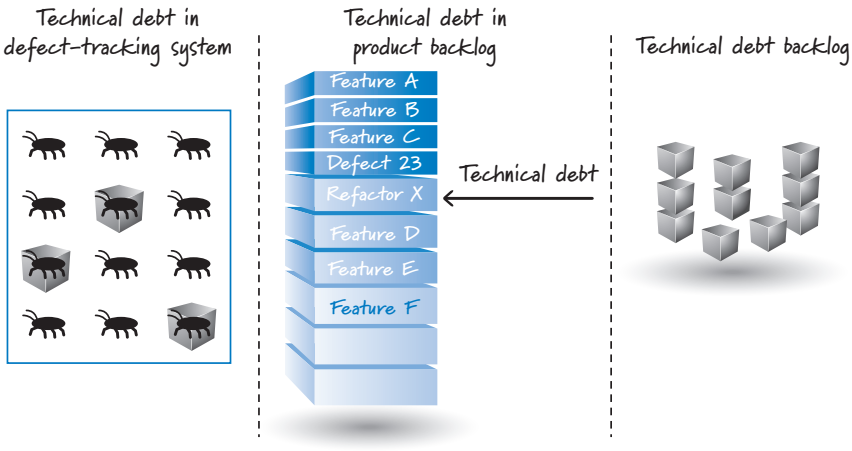

FIGURE 8.9 Ways to make technical debt visible at the technical level

because the team may choose to service the debt differently from the way it services defects (as I will discuss shortly).

Another approach to making technical debt visible is to create product backlog items that represent technical debt (middle of Figure 8.9). Doing so will give important technical debt visibility on a par with that of new features in the product backlog. Teams typically use this approach when the cost of servicing the technical debt is quite high and the product owner needs to be involved in deciding how that work should be ordered relative to value-added new features in the product backlog.

A third approach to making technical debt visible is to create a special technical debt backlog that makes individual technical debt items visible (right side of Figure 8.9). Whenever new technical debt is discovered or introduced into the product, a development team member can create a new technical debt item and add it to the technical debt backlog. By making the technical debt items visible, the development team can not only see its technical debt position but also can proactively determine when it wants to service each piece of technical debt.

For colocated teams, a simple approach to visualizing the technical debt backlog is to create a technical debt board on the wall and use sticky notes or cards to represent specific technical debt items. Usually the technical debt board would be placed right next to the sprint backlog so that during sprint planning the team has visibility into the technical debt that it can consider servicing in the upcoming sprint (I will discuss this approach in the next section).

Most teams treat the technical debt backlog in a low-ceremony way by just placing technical debt cards on the wall. However, others might choose to groom the technical debt backlog a bit more by investing a little time to order the cards or to give a rough idea of the effort required to address the debt described on the card.

Servicing the Technical Debt

The last activity in managing technical debt is to service or repay the debt. When discussing debt servicing, I find it helpful to use the following status categories:

- **Happened-upon technical debt**—debt that the development team was unaware existed until it was exposed during the normal course of performing work on the product. For example, the team is adding a new feature to the product and in doing so it realizes that a work-around had been built into the code years before by someone who has long since departed.
- **Known technical debt**—debt that is known to the development team and has been made visible using one of the previously discussed approaches.
- **Targeted technical debt**—debt that is known and has been targeted for servicing by the development team.

Based on these categories, I generally apply the following algorithm when servicing technical debt:

1. Determine if the known technical debt should be serviced (as I will discuss, not all debt should be serviced). If it should be serviced, go to step 2.
2. If you are in the code doing work and you discover happened-upon technical debt, clean it up. If the amount of happened-upon technical debt exceeds some reasonable threshold, clean it up until you reach that threshold. Then classify the nonserviced, happened-upon technical debt as known technical debt (for example, by creating entries in the technical debt backlog).
3. Every sprint, consider designating some amount of known technical debt as targeted technical debt to be serviced during the sprint. Favor servicing known technical debt with a high interest rate that is aligned with customer-valuable work.

The approaches shown in Figure 8.10 expand on this algorithm for servicing technical debt.

I will describe each of these approaches and how they specifically apply when using Scrum.

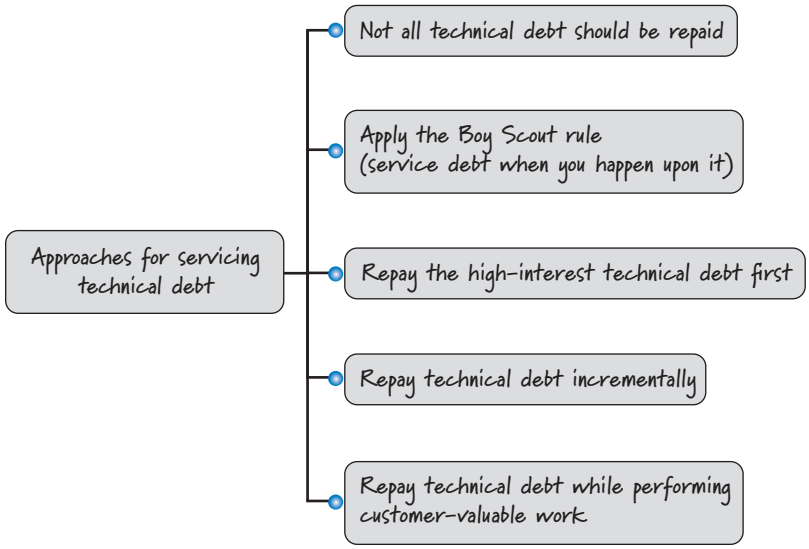

FIGURE 8.10 Approaches for servicing technical debt

Not All Technical Debt Should Be Repaid

Sometimes technical debt should not be repaid. This is one area where the analogy with financial debt gets stretched. Typically the expectation is that all financial debt eventually will be repaid—although we know that isn't always true!

There are a number of scenarios under which technical debt should not be repaid. I will discuss three: product nearing end of life, throwaway prototype, and product built for a short life.

Product Nearing End of Life

If a product has accrued significant technical debt and it is approaching end of life, investing in any substantial debt repayment would be fiscally irresponsible. If the product is low value, we would likely retire the product (and therefore the debt) and devote our resources to higher-value products. If we have a high-value, high-technical-debt product that is nearing end of life, it might make more sense to take on the high risk and high cost of developing a new product than to repay the technical debt in the old product.

Throwaway Prototype

There are times when deliberately taking on technical debt with absolutely no plan to ever repay it might be the most economically sensible thing to do. A common example would be the development of a throwaway prototype that is created for knowledge-acquisition purposes (Goldberg and Rubin 1995). The value of the prototype is not the code but rather the validated learning we get (Ries 2011). Because the prototype is not engineered for a life in the marketplace, it likely has some or a lot of technical debt. However, because it is a throwaway prototype, there is no reason to repay the debt. Of course, if we create a throwaway prototype and then decide not to throw it away, but instead treat it like an evolutionary prototype and evolve it into the product, we will almost certainly be starting with a foundation that is mired in significant technical debt.

Product Built for a Short Life

If we build a product for a very short production life, the economics might dictate that technical debt should not be repaid. I illustrate this scenario with an interesting example that I encountered in the late 1980s. At the time I was working for ParcPlace Systems, the early market leader in object-oriented development environments. Back in those days I was helping several high-profile Wall Street banks adopt Smalltalk as a development platform. In one particular case I was brought in to coach a team to help its members better understand object-oriented technology and to more effectively use the Smalltalk development environment. This team had just produced one of the first-of-its-kind derivative trading systems. When I arrived, one of my first

requests to the group VP was to review the design and implementation of the product the team had just built—the product had not yet gone live but was scheduled to do so soon.

After a day of reviewing the architecture and code I met with the VP and told him that his system might just be the nastiest-looking Smalltalk implementation I had ever seen. I pointed out that the implementation had enormous problems that had to be addressed immediately or their system (and business) was in for a world of hurt.

At that point the VP told me (word for word), "Son, if you spend one nickel to clean up that system, I will personally take you out back and shoot you." I was, to say the least, dumbfounded by his remark. I responded, "You need to trust me on this one. That system is poorly designed and horribly implemented and you will have long-term problems with it." He retorted, "You don't understand my business. In my marketplace, when we come out with a new financial instrument, we make the lion's share of our profit in the first three months. That's about how long it takes for my competitors to rush in with their 'me-too' products. At that point, I am better off exiting that market and developing a new product. I only need that new system to last three months. I don't care if you hold it together with chewing gum and baling wire. Just don't delay my revenue generation and give my competitors an opportunity to beat me to market. We're turning it on."

That's exactly what they did. In the first *hour* the system was in operation, the traders using it generated $14M in revenue. I personally thought they took a large risk by turning the system on in its fragile state, but from a revenue perspective I was wrong.

Usually organizations don't build products with an expected life of three months. Typically we are interested in engineering a product for an extended life in the marketplace.

Apply the Boy Scout Rule (Service Debt When You Happen Upon It)

There is a **Boy Scout rule**: "Always leave the campground cleaner than you found it." If you find a mess on the ground, you clean it up regardless of who might have made the mess. You intentionally improve the environment for the next group of campers. Bob Martin (and others) has nicely explained why this rule applies to product development and technical debt (Martin 2008).

Following this rule, we try to always make our product design and implementation a little better, not a little worse, every time we touch it. When a development team member is working in an area of the product and sees a problem (happened-upon technical debt), she cleans up the problem. She doesn't do this just because it is good for her, though it almost certainly is, but also because it is good for the whole development team and the organization.

The algorithm provided earlier stated that we service the happened-upon debt up to a reasonable threshold. We can't just blatantly say that the team should service the entire happened-upon technical debt when it is discovered. The servicing of that debt might require significant effort, and the team is in the middle of a sprint in which it has other work to complete. If the team tries to service the entire debt, it may not be able to meet its original sprint goal.

To address this issue, the team might budget a percentage of time to allow for servicing happened-upon debt when it is discovered. One way to set this budget is to increase the estimated size of individual PBIs to allow for the additional debt servicing that typically occurs. Alternatively, the team might choose to budget a percentage of its capacity during sprint planning to service happened-upon debt. Examples I have seen in the past range from 5% up to 33% of the sprint capacity. You should let your particular circumstances guide your capacity allocation if you choose to use this approach.

As for any happened-upon debt that is not serviced when discovered, it should be classified as known debt and made visible using whatever technique the team has decided to use for visualizing technical debt.

Repay Technical Debt Incrementally

In some products the accrued technical debt level might be quite high. Teams working on such products frequently end up making large balloon payments as a means of servicing their debt load. They would be far better off if they made many, timely, incremental payments against known technical debt instead of large late payments. Smaller, more frequent payments are akin to making monthly payments against a home mortgage. Doing so allows some of the debt to be serviced each month, avoiding a large balloon payment at the end of the loan.

I get concerned when I hear teams discussing their "technical debt sprints" or "refactoring sprints." These are sprints whose only goal is to perform technical debt reduction work. These sound like balloon payments to me. In fact, these sprints give the appearance that the debt level was allowed to grow without attention to reduction. Now it has become such a problem that instead of developing customer-valuable features in the next sprint, the team is going to deliver no customer value but instead dedicate itself to dealing with a problem that it should have been dealing with a little bit each sprint. There are times when the technical debt is so high and attention to it so low that a sprint dedicated to raising awareness and making a concerted, full-team-focused effort on repayment is helpful. However, as a rule, such sprints are to be avoided whenever possible; repayment should occur incrementally.

Using this approach, we take some amount of known technical debt and designate it as targeted technical debt to be serviced during the next sprint. The decision as to how much targeted technical debt to take on each sprint can be made by the Scrum team during sprint planning.

Repay the High-Interest Technical Debt First

Although it is convenient to lump all types of shortcuts or deficiencies under one label of technical debt, it is important to realize that not all types of technical debt are of equal importance. An example of an important form of debt is a frequently modified module that a lot of other code depends on and is in real need of refactoring because it's becoming increasingly difficult to change. We pay interest on that debt all of the time, and the magnitude of the interest continues to increase as we make more and more changes.

On the other hand, we could have technical debt (known design or implementation issues) in a part of the product that is rarely used and almost never modified. On a day-to-day basis we are not paying any, or at least not much, interest on this debt. This is not a form of debt that requires a lot of attention, unless there is a not-so-insignificant risk that this part of the product could fail and that the failure would have major repercussions.

When servicing technical debt, therefore, we should target and service the high-interest technical debt first. Any reasonable business person would do the same with financial debt. For example, unless there is a compelling reason, as a rule we would pay off the financial debt with an 18% interest rate before we repay the debt with a 6% interest rate.

Some organizations have accrued such high levels of technical debt that they can become a bit paralyzed because they don't know how to get started. For them, the high-interest debt might be obvious but daunting in size. To prime the pump of debt reduction, they may choose to repay a small debt to get accustomed to the process of debt repayment. I am in favor of taking whatever actions might be culturally necessary to give organizations the jolt they need to start managing their debt. As I will describe next, if we repay technical debt while performing customer-valuable work, we can incrementally focus on a small amount of debt that is worth repaying.

Repay Technical Debt While Performing Customer-Valuable Work

An excellent way to repay known technical debt incrementally, while focusing on high-interest technical debt and aligning technical debt servicing with the Scrum value-centric approach, is to make debt payments while performing customer-valuable work. So, whenever possible, avoid scheduling a full sprint of debt reduction work, or for that matter defining individual product backlog items that are specific to debt reduction. Instead, we should service known technical debt coincident with the development of customer-valuable features in the product backlog.

Let's assume that for every customer-valuable product backlog item we work on, we also do several things. First, we commit to doing high-quality work so that we don't add new naive technical debt when we create the customer feature. Second, we apply the Boy Scout rule and clean up whatever happened-upon technical debt that we reasonably can when we are in the area doing work related to our feature. And

third (the core attribute of this approach), we specifically repay targeted technical debt in the area in which we will be working.

Using this approach has several advantages:

- It aligns debt reduction work with customer-valuable work that the product owner can properly prioritize.
- It makes it clear to all development team members that technical debt reduction is a shared responsibility and not something to defer and delegate to someone or some other team to clean up.
- It reinforces technical debt prevention and removal skills because everyone gets to practice them all the time.
- It helps us identify the high-interest areas where we should focus our technical debt servicing. At the very least we know that the code (or other development artifact) we are touching is still important because we are using it to create the new feature.
- It avoids the waste of repaying technical debt in areas where we really don't have to.

Earlier I mentioned an approach that I have seen several Scrum teams use to help manage the alignment of known technical debt reduction activities with product backlog items (shown in Figure 8.11).

Using this approach, known technical debt items are entered into a technical debt backlog that is placed on the wall next to the sprint backlog during sprint planning (or inside a tool to achieve the equivalent effect).

During sprint planning, as the team members are working with the product owner to select customer-valuable items from the product backlog to work on in the next sprint, they consider the cards on the technical debt board to see if the work they are planning to do on the new product backlog item would naturally intersect an area of the product associated with a technical debt card. If so, someone takes the card from the technical debt board and places it in the sprint backlog as work for this sprint. Then, when performing the work necessary to complete the product backlog

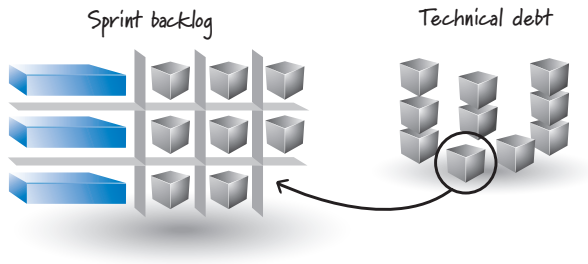

FIGURE 8.11 A technique for managing technical debt when using Scrum

item, the team members would also address the technical debt tasks they pulled into the sprint.

This approach is a very simple and elegant way of aligning technical debt servicing with the creation of user value.

Closing

In this chapter I discussed the concept of technical debt, which accrues when we take shortcuts today at tomorrow's expense. I distinguished among naive, unavoidable, and strategic technical debt. I went on to explain the consequences of poorly managed levels of technical debt. I then discussed the three activities associated with controlling technical debt: managing the accrual of technical debt, making technical debt visible, and servicing technical debt.

This chapter concludes Part I. In the next chapter, I will transition and begin the discussion of the various roles on a Scrum development effort, beginning with the product owner role.